Contributing to SolveSpace

==========================

Contributing bug reports

------------------------

Bug reports are always welcome! When reporting a bug, please include the following:

\* The version of SolveSpace (use Help ? About...);

\* The operating system;

\* The save file that reproduces the incorrect behavior, or, if trivial or impossible,

instructions for reproducing it.

GitHub does not allow attaching `\*.slvs` files, but it does allow attaching `\*.zip` files,

so any savefiles should first be archived.

Signing the CLA

---------------

To contribute code, translations, artwork, or other resources to SolveSpace, it is necessary to

sign a [Contributor License Agreement](https://cla-assistant.io/solvespace/solvespace).

Contributing translations

-------------------------

To contribute a translation, not a lot is necessary—at a minimum, you need to be able

to edit .po files with a tool such as [poedit](https://poedit.net/). Once you have

such a tool installed, take `res/messages.pot` and start translating!

However, if you want to see your translation in action, a little more work is necessary.

First, you need to be able to build SolveSpace; see [README](README.md). After that:

\* Copy `res/messages.pot` to `res/locales/xx\_YY.po`, where `xx` is an ISO 639-1

country code, and `YY` is an ISO 3166-1 language code.

\* Add a line `xx-YY,LCID,Name` to `res/locales.txt`, where `xx-YY` have the same

meaning as above, `LCID` is a Windows Language Code Identifier ([MS-LCID][]

has a complete list), and `Name` is the full name of your locale in your language.

\* Add `locales/xx\_YY.po` in `res/CMakeLists.txt`—search for `locales/en\_US.po`

to see where it should be added.

You're done! Recompile SolveSpace and you should be able to select your translation

via Help ? Language.

[MS-LCID]: https://msdn.microsoft.com/en-us/library/cc233965.aspx

Contributing code

-----------------

SolveSpace is written in C++, and currently targets all compilers compliant with C++11.

This includes GCC 5 and later, Clang 3.3 and later, and Visual Studio 12 (2013) and later.

### High-level conventions

#### Portability

SolveSpace aims to consist of two general parts: a fully portable core, and platform-specific

UI and support code. Anything outside of `src/platform/` should only use standard C++11,

and rely on `src/platform/unixutil.cpp` and `src/platform/w32util.cpp` to interact with

the OS where this cannot be done through the C++11 standard library.

#### Libraries

SolveSpace primarily relies on the C++11 STL. STL has well-known drawbacks, but is also

widely supported, used, and understood. SolveSpace also includes a fair amount of use of

bespoke containers List and IdList; these provide STL iterators, and can be used when

convenient, such as when reusing other code.

One notable departure here is the STL I/O threads. SolveSpace does not use STL I/O threads

for two reasons: (i) the interface is borderline unusable, and (ii) on Windows it is not

possible to open files with Unicode paths through STL.

When using external libraries (other than to access platform features), the libraries

should satisfy the following conditions:

\* Portable, and preferably not interacting with the platform at all;

\* Can be included as a CMake subproject, to facilitate Windows, Android, etc. builds;

\* Use a license less restrictive than GPL (BSD/MIT, Apache2, MPL, etc.)

#### String encoding

Internally, SolveSpace exclusively stores and uses UTF-8 for all purposes; any `std::string`

may be assumed to be encoded in UTF-8. On Windows, UTF-8 strings are converted to and from

wide strings at the boundary; see [UTF-8 Everywhere][utf8] for details.

[utf8]: http://utf8everywhere.org/

#### String formatting

For string formatting, a wrapper around `sprintf`, `ssprintf`, is used. A notable

pitfall when using it is trying to pass an `std::string` argument without first converting

it to a C string with `.c\_str()`.

#### Filesystem access

For filesystem access, the C standard library is used. The `ssfopen` and `ssremove`

wrappers are provided that accept UTF-8 encoded paths.

#### Assertions

To ensure that internal invariants hold, the `ssassert` function is used, e.g.

`ssassert(!isFoo, "Unexpected foo condition");`. Unlike the standard `assert` function,

the `ssassert` function is always enabled, even in release builds. It is more valuable

to discover a bug through a crash than to silently generate incorrect results, and crashes

do not result in losing more than a few minutes of work thanks to the autosave feature.

### Use of C++ features

The conventions described in this section should be used for all new code, but there is a lot

of existing code in SolveSpace that does not use them. This is fine; don't touch it if it works,

but if you need to modify it anyway, might as well modernize it.

#### Exceptions

Exceptions are not used primarily because SolveSpace's testsuite uses measurement

of branch coverage, important for the critical parts such as the geometric kernel.

Every function call with exceptions enabled introduces a branch, making branch coverage

measurement useless.

#### Operator overloading

Operator overloading is not used primarily for historical reasons. Instead, method such

as `Plus` are used.

#### Member visibility

Member visibility is not used for implementation hiding. Every member field and function

is `public`.

#### Constructors

Constructors are not used for initialization, chiefly because indicating an error

in a constructor would require throwing an exception, nor does it use constructors for

blanket zero-initialization because of the performance impact of doing this for common

POD classes like `Vector`.

Instances can be zero-initialized using the aggregate-initialization syntax, e.g. `Foo foo = {};`.

This zero-initializes the POD members and default-initializes the non-POD members, generally

being an equivalent of `memset(&foo, 0, sizeof(foo));` but compatible with STL containers.

#### Input- and output-arguments

Functions accepting an input argument take it either by-value (`Vector v`) or

by-const-reference (`const Vector &v`). Generally, passing by-value is safer as the value

cannot be aliased by something else, but passing by-const-reference is faster, as a copy is

eliminated. Small values should always be passed by-value, and otherwise functions that do not

capture pointers into their arguments should take them by-const-reference. Use your judgement.

Functions accepting an output argument always take it by-pointer (`Vector \*v`). This makes

it immediately visible at the call site as it is seen that the address is taken. Arguments

are never passed by-reference, except when needed for interoperability with STL, etc.

#### Iteration

`foreach`-style iteration is preferred for both STL and `List`/`IdList` containers as it indicates

intent clearly, as opposed to `for`-style.

#### Const correctness

Functions that do not mutate `this` should be marked as `const`; when iterating a collection

without mutating any of its elements, `for(const Foo &elem : collection)` is preferred to indicate

the intent.

### Coding style

Code is formatted by the following rules:

\* Code is indented using 4 spaces, with no trailing spaces, and lines are wrapped

at 100 columns;

\* Braces are placed at the end of the line with the declaration or control flow statement;

\* Braces are used with every control flow statement, even if there is only one statement

in the body;

\* There is no space after control flow keywords (`if`, `while`, etc.);

\* Identifiers are formatted in camel case; variables start with a lowercase letter

(`exampleVariable`) and functions start with an uppercase letter (`ExampleFunction`).

For example:

```c++

std::string SolveSpace::Dirname(std::string filename) {

int slash = filename.rfind(PATH\_SEP);

if(slash >= 0) {

return filename.substr(0, slash);

}

return "";

}

```

If you install [clang-format][], this style can be automatically applied by staging your changes

with `git add -u`, running `git clang-format`, and staging any changes it made again.

[clang-format]: https://clang.llvm.org/docs/ClangFormat.html

Debugging code

--------------

SolveSpace releases are thoroughly tested but sometimes they contain crash

bugs anyway. The reason for such crashes can be determined only if the executable

was built with debug information.

### Debugging a released version

The Linux distributions usually include separate debug information packages.

On a Debian derivative (e.g. Ubuntu), these can be installed with:

apt-get install solvespace-dbg

The macOS releases include the debug information, and no further action

is needed.

The Windows releases include the debug information on the GitHub

[release downloads page](https://github.com/solvespace/solvespace/releases).

### Debugging a custom build

If you are building SolveSpace yourself on macOS, use the XCode

CMake generator, then open the project in XCode as usual, select

the Debug build scheme, and build the project:

cd build

cmake .. -G Xcode [other cmake args...]

If you are building SolveSpace yourself on any Unix-like platform,

configure or re-configure SolveSpace to produce a debug build, and

then build it:

cd build

cmake .. -DCMAKE\_BUILD\_TYPE=Debug [other cmake args...]

make

If you are building SolveSpace yourself using the Visual Studio IDE,

select Debug from the Solution Configurations list box on the toolbar,

and build the solution.

### Debugging with gdb

gdb is a debugger that is mostly used on Linux. First, run SolveSpace

under debugging:

gdb [path to solvespace executable]

(gdb) run

Then, reproduce the crash. After the crash, attach the output in

the console, as well as output of the following gdb commands to

a bug report:

(gdb) backtrace

(gdb) info locals

If the crash is not easy to reproduce, please generate a core file,

which you can use to resume the debugging session later, and provide

any other information that is requested:

(gdb) generate-core-file

This will generate a large file called like `core.1234` in the current

directory; it can be later re-loaded using `gdb --core core.1234`.

### Debugging with lldb

lldb is a debugger that is mostly used on macOS. First, run SolveSpace

under debugging:

lldb [path to solvespace executable]

(lldb) run

Then, reproduce the crash. After the crash, attach the output in

the console, as well as output of the following gdb commands to

a bug report:

(lldb) backtrace all

(lldb) frame variable

If the crash is not easy to reproduce, please generate a core file,

which you can use to resume the debugging session later, and provide

any other information that is requested:

(lldb) process save-core "core"

This will generate a large file called `core` in the current

directory; it can be later re-loaded using `lldb -c core`.

### Debugging GUI-related bugs on Linux

There are several environment variables available that make crashes

earlier and errors more informative. Before running SolveSpace, run

the following commands in your shell:

export G\_DEBUG=fatal\_warnings

export LIBGL\_DEBUG=1

export MESA\_DEBUG=1